Comparing Formalizations of Proofs about Programming Languages

Yanjun Yang, 2020

Advisors: David Walker, Matthew Weaver

April 27, 2019

Abstract

Type safety is an important property to have for typed languages because it guarantees well-defined evaluation semantics for certain terms in the language without additional proof from a user. However, proofs of type safety can be complex even for simple languages. In this project, we compared three different formalizations of the simply typed lambda calculus ($\lambda^{\rightarrow}$) to determine how certain language features, such as the use of de Brujin indices to reference variables and/or the use of intrinsic typing, affect the structure and complexity of type-safety proofs. We found that using de Brujin indices can significantly reduce the complexity of type safety proofs because no extra work is needed to properly define capture-avoiding substitution and variable shadowing. We also found that while using intrinsic typing only marginally reduces the length of the type-safety proof, it makes the overall proof more elegant because it removes the need to define and prove duplicate claims about terms and their typing judgments. However, we find a language that uses de Brujin indices and intrinsic typing is much more difficult for humans to interpret.

1 Introduction

As we continue into the 21st century, we start to see that computer programming begins to take on a larger and larger role in society. We begin to see that more and more parts of our world are rapidly becoming automated by computers and machines, and as our reliance on technology continues to increase, so does the need to verify that the computer programs behind such technology is robust and behaves in ways that had been intended.

One way in which we can verify the correctness of our programs is through testing them using a variety of inputs, many times with adversarial ones, to determine whether they exhibit the desired behaviors or not. However, this method is not fool-proof, as it is likely the case for most real world applications that exhaustively testing all possible inputs is practically unfeasible. Thus, we turn to mathematical proofs of correctness. The idea behind this is to use logic to deduce the correctness of our programs. Unfortunately, this can be difficult to accomplish on a per-program basis, but on the other hand, if we were to prove certain properties about the underlying programming language, then we can automatically guarantee that all programs written in that language would have those properties, which is very desirable.

One property of programming languages that is often explored is **type safety**. Type safety is the property that guarantees that terms in the language that are well typed will also have well-defined evaluation semantics. This means that terms, and in general, programs, that are shown to have a type will never raise an error or crash during execution. In essence, a term that is well-typed is automatically endowed with a certain degree of correctness. While one can easily see why type safety is a desirable property to have for any programming language, proving that a language is type safe can be complicated even for the simplest of programming languages.

In this project, we will be examining a single language, the **simply typed lambda calculus**, in detail and comparing how different formalizations of the same language can affect the corresponding proofs of type safety. The goal is to determine what features of programming languages can make type-safety easier to prove formally and verify using proof assistants.

2 Background and Related Work

Much work has been done previously on the subject of types and type safety. The first type-safe programming language was the simply typed lambda calculus, created by Alonzo Church in 1940 [1], where its type-safety allowed it to be used as a model for constructive logics. This language is also typically referred to as $\lambda^{\rightarrow}$. Later on, more advanced type theories which use dependent types were developed, such as Per Martin-Löf's *Intuitionistic Type Theory* in 1973 [5]. Dependent types are defined as any type that is parameterized by terms of another type. More recently, the Univalent Foundations Program's Homotopy Type Theory (HoTT) in 2013 was developed [8], and Cohen, Coquand, Huber, and Mörtberg developed a constructive formulation of HoTT known as Cubical Type Theory in 2016 [2]. On the other hand, type-safety proofs have also been studied extensively. *Types and Programming Languages*, written by Benjamin Pierce in 2002, lays out much of the theoretical foundation for programming language theory, which, among other topics, discusses the general outline of proofs of type safety [6].

In this project, we used $\lambda^{\rightarrow}$, as defined by Church [1], as the object of study. We examined different formalizations of $\lambda^{\rightarrow}$ and compared the ways in which they affect the corresponding proofs of type safety, which we prove using the techniques described in *Types and Programming Languages* [6].

3 Implementation

3.1 Proof Assistant: Agda

To carry out the proofs of type safety, we used the proof assistant/programming language Agda to formally define the abstract syntax of $\lambda^{\rightarrow}$ and to formally prove type safety thereof. Agda is a dependently-typed functional programming language based primarily on Martin-Löf's intuitionistic type theory [3]. As such, it is able to encode a constructive predicate logic as programming language through the equivalence of proofs and programs, commonly known as the Curry-Howard isomorphism [4]. In particular, logical implications are encoded as functions, universal quantifications are encoded as dependently-typed functions, and existential quantifications are encoded as dependently-typed pairs.

In Agda, new data types are defined inductively. For example, one can define the natural numbers as:

```
data Nat : Set where
    Zero : Nat
    Suc : Nat -> Nat
```

and one can write down terms of type Nat by using the inductively-defined constructors as such:

```
One : Nat                              Two : Nat
One = Suc Zero                         Two = Suc (Suc Zero)
```

and so forth. Now, suppose that we have a predicate $P(\cdot)$, which is indexed by natural numbers. To prove $P(n)$ for all natural numbers n, we would write a dependently-typed function that takes natural numbers n and sends them to proofs of $P(n)$. The body of this function would be defined inductively, which mirrors a proof by induction of $P(n)$ over the natural numbers. In Agda, we get:

```
P : Nat -> Set                         P-Proof : (n : Nat) -> P n
P = ...                                P-Proof Zero = ...
                                       P-Proof (Suc n') = ...
                                           -- but can use P-Proof n'
```

Using these methods, we encoded the proofs that we carried out in this project.

3.2 Simply Typed Lambda Calculus

As mentioned before, the object of study for this project is the simply typed lambda calculus ($\lambda^{\rightarrow}$) [1]. We will also be endowing $\lambda^{\rightarrow}$ with a base type bool, as well as base terms true and false. The abstract syntax of $\lambda^{\rightarrow}$ can be defined as a context-free grammar as follows:

$$e ::= \text{true} \mid \text{false} \mid x \mid \lambda x : \tau . e \mid e\ e$$
$$\tau ::= \text{bool} \mid \tau \rightarrow \tau$$

The first line defines the terms e in the language, which can be one of five different expressions: the constant true, the constant false, a variable, a function (also called a λ-abstraction), and an application. The function binds a type τ for its x, and the types τ in the language are defined on the second line as either the base type bool, or a function type. For example, we can write down the identity function on booleans as follows:

$$\lambda x : \text{bool}.x$$

and we can write the identity function on booleans applied to the boolean true as:

$$(\lambda x : \text{bool}.x)\ \text{true}$$

So far, we have seen that variables bound by functions are given a type: the same type that is bound by that function. This idea of typedness can be extended to terms in the language as well. Formally, the type of a term can be defined through the following deduction rules:

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}}\ \text{T-True} \qquad \frac{}{\Gamma \vdash \text{false} : \text{bool}}\ \text{T-False} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \text{T-Var}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau.e : \tau \to \tau'}\ \text{T-Fun} \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'}\ \text{T-App}$$

where Γ, the context, is defined as:

$$\Gamma ::= \varnothing \mid \Gamma, x : \tau$$

T-True and T-False axiomatically define the type of true and false to be bool in any context Γ. T-Var types a variable with the type it is given in the context Γ. T-Fun types a function as $\tau \to \tau'$ if the variable has type τ and the body of the function has type τ' in the context with the said variable. Finally, T-App types a function application as τ' if the function has type $\tau \to \tau'$ and the argument that is applied has type τ.

In this project, we defined evaluation ($\longrightarrow$) of the $\lambda^{\to}$ to use call-by-value order, which is captured by the following rules:

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}\ \text{E-App1} \qquad \frac{e_2 \longrightarrow e_2'}{v_1\ e_2 \longrightarrow v_1\ e_2'}\ \text{E-App2} \qquad \frac{\Gamma \vdash \lambda x : \tau.e : \tau \to \tau' \qquad \Gamma \vdash v : \tau}{(\lambda x : \tau.e)\ v \longrightarrow [v/x]\ e}\ \text{E-AppFun}$$

where v, the values in the language (i.e. the terms that cannot be evaluated further), are the constants true and false, as well as functions that contain no unbound variables, and $[v/x]\ e$ is the substitution function, which replaces all unbound instances of the variable x in e with the value v. By convention, the substitution function is also capture avoiding, which implies substitution will not change the semantics of unbound variables in the substituting term v. However, values, as we have defined them, will automatically have no unbound variables, so all substitutions are automatically capture avoiding.

3.3 Type Safety

As explained by *Types and Programming Languages*, proofs of type safety are typically carried out in two parts, conventionally known as **Progress** and **Preservation**, respectively. These theorems claim the following:

Theorem 1. *(Progress) For all terms e, if e is well typed, then either e is a value, or there exists some e' such that $e \longrightarrow e'$.*

Theorem 2. *(Preservation) For all terms e and e', if e is well typed and $e \longrightarrow e'$, then e' is well typed and has the same type as e.*

Composing the theorems, we get that if a term e is well typed, then either it cannot be evaluated further, or it evaluates to another well-typed term of the same type, which means that these theorems apply again to the new term. In essence, these theorems guarantee that all well-typed terms have well-defined evaluation semantics.

3.4 formalizations of $\lambda^{\rightarrow}$ in Agda

3.4.1 Extrinsically Typed $\lambda^{\rightarrow}$ with Named Variables

The first formalization of $\lambda^{\rightarrow}$ that we will explore uses **named variables** and **extrinsic typing**. Named variables are variables which refer to the arguments bound by functions by the name they are bound with. For example, consider the identity function on booleans:

$$\lambda x : \text{bool}.x$$

The variable in the body of the function refers to the argument of the function with the name x because x is what the function binds as its argument. To formalize this in Agda, we first define a name type, which we arbitrarily choose to use natural numbers in the underlying representation:

```
data Name : Set where
  N : Nat -> Name
```

We then continue on to define the data types for types and terms in the language:

```
data Type : Set where
  Boolean : Type
  Function : Type -> Type
    -> Type
```

```
data Term : Set where
  True : Term
  False : Term
  Var : Name -> Term
  Fun : Name -> Type -> Term
    -> Term
  App : Term -> Term -> Term
```

Conversely, extrinsic typing is when the type of the term, along with the mechanisms for determining the type, are external to the term itself. Therefore, we will need additional data types to encode the typing judgments. We now define the Context data type as a list of pairs of names and types, as follows:

```
Context : Set
Context = List (Name × Type)
```

This allows us to define the Type-Proof data type, which encodes our five typing judgments, as the following dependent type:

```
data Type-Proof (Γ : Context) : Term -> Type -> Set where
  Type-True : Type-Proof Γ True Boolean
  Type-False : Type-Proof Γ False Boolean
  Type-Var : (n : Name) (t : Type) (p : (n , t) ∈ Γ)
    -> Type-Proof Γ (Var n) t
  Type-Fun : (n : Name) (t t' : Type) (e : Term)
    -> Type-Proof ((n , t) :: Γ) e t'
```

```
        -> Type-Proof Γ (Fun n t e) (Function t t')
    Type-App : (t t' : Type) (e1 e2 : Term)
        -> Type-Proof Γ e1 (Function t t')
        -> Type-Proof Γ e2 t
        -> Type-Proof Γ (App e1 e2) t'
```

Finally, we must encode our values and our evaluation semantics. The details of the definition are not as relevant, so only the types and constructor names will be shown here:

```
data IsVal-Proof : Term -> Set where   data Execution-Proof : Term -> Term
    IsVal-True : ...                           -> Set where
    IsVal-False : ...                          Execution-App1 : ...
    IsVal-Fun : ...                            Execution-App2 : ...
                                               Execution-AppFun : ...
```

For the full definition, please see the Appendix. To prove type safety, we must prove the Progress and Preservation theorems, which we encode as follows:

```
Progress : (e : Term) (t : Type) -> Type-Proof [] e t
    -> IsVal-Proof e ⊎ Σ [e' ∈ Term] Execution-Proof e e'
Preservation : (e e' : Term) (t : Type) -> Type-Proof [] e t
    -> Execution-Proof e e' -> Type-Proof [] e' t
```

To prove Progress, we must define a function that takes in a term `e`, a type `t`, and a proof that `e` has type `t`, and returns either a proof that `e` is a value or that `e` evaluates to some `e'`. To prove Preservation, we must define a function that takes in two terms `e` and `e'`, a type `t`, a proof that `e` has type `t`, and a proof that `e` evaluates to `e'`, and returns a proof that `e'` has type `t`.

3.4.2 Extrinsically Typed $\lambda^{\rightarrow}$ with Nameless Variables

Next, we formalize $\lambda^{\rightarrow}$ using extrinsic typing, but **nameless variables**. In particular, we will be referring to a variable via its **de Brujin index**. The de Brujin index of a variable is the number of additional argument bound between where the referencing expression and the referenced variable. For example, consider the following lambda term:

$$\lambda x : \text{bool} \rightarrow \text{bool}. x \ ((\lambda y : \text{bool}. x \ y) \ \text{true})$$

In nameless representation, this becomes:

$$\lambda\_\_ : \text{bool} \rightarrow \text{bool}. \langle 0 \rangle \ ((\lambda\_\_ : \text{bool}. \langle 1 \rangle \ \langle 0 \rangle) \ \text{true})$$

In the inner function, we see that the de Brujin index of the variable y is 0 because it is bound immediately preceding where it is used, and the de Brujin index of x is 1 because y is bound before x is used here. However, outside the inner function, we see that the variable x has de Brujin index 0, since y is not yet bound.

The advantage of referring to a variable by its de Brujin index is that there is no ambiguity as to which variable is being referenced. For example, if we have the following expression using named variables:

$$\lambda x : \text{bool} \rightarrow \text{bool}. \lambda x : \text{bool} \rightarrow \text{bool}. x \ \text{true}$$

the x in the expression could be interpreted to refer to either the outer or the inner bound x. Conventionally, we take this to refer to the inner x. However, with de Brujin indices, we get:

$$\lambda\_\_ : \text{bool} \rightarrow \text{bool}. \lambda\_\_ : \text{bool} \rightarrow \text{bool}. \langle 0 \rangle \ \text{true}$$

where it is completely unambiguous that we are referring to the inner argument.

To encode this formalization in Agda, we do not need to modify the data type of types, but we must redefine the data type of terms. But to do so, we must first define the de Brujin index representation of variables, which we do as follows:

```
data Type-Box : Type -> Set where
    Box : (t : Type) -> Type-Box t
data Context : Set where
    Empty : Context
    _,_ : Context -> Type
        -> Context
```

```
Variable : Context -> Set
Variable Empty = ⊥
Variable (Γ , t) = (Variable Γ)
    ⊎ (Type-Box t)
```

Note that contexts no longer include variable names. The above encoding allows us to write down contexts and de Brujin indices as follows: Suppose we have the following context:

```
Γ : Context
Γ = (Boolean , Function Boolean Boolean) , Boolean
```

To reference the variables in this context, we use the following expressions:

```
Variable Γ : Set
Variable Γ = (Type-Box Boolean) ⊎ (Type-Box (Function Boolean Boolean))
    ⊎ (Type-Box Boolean)
Var-Zero : Variable Γ
Var-Zero = inr (Box Boolean)
Var-One : Variable Γ
Var-One = inl (inr (Box (Function Boolean Boolean)))
Var-Two : Variable Γ
Var-Two = inl (inl (inr (Box Boolean)))
```

where `inl` and `inr` are the constructors of the sum type ⊎. We see that this is analogous to how we define the natural numbers, where `inr (...)` corresponds to `Zero` and `inl` corresponds to `Suc`.

Now, we can define the terms in the language as follows:

```
data Term (Γ : Context) : Set where
    True : Term Γ
    False : Term Γ
    Var : Variable Γ -> Term Γ
    Fun : (t : Type) -> Term (Γ , t) -> Term Γ
    App : Term Γ -> Term Γ -> Term Γ
```

Note that in the above formalization, the body of a function must be a term in an extended context, since it is allowed to refer to the variable bound by the function in addition to all other variables already available.

The typing judgment, values, and evaluation semantics are analogous to before, so they will not be shown here. For the full definitions, please see the Appendix.

The Progress and Preservation theorems for this formalization of $\lambda^{\rightarrow}$ can be encoded as:

```
Progress : (e : Term Empty) (t : Type) -> Type-Proof Empty e t
    -> IsVal-Proof e ⊎ Σ [e' ∈ Term Empty] Execution-Proof e e'
Preservation : (e : Term Empty) (t : Type) (e' : Term Empty)
    -> Type-Proof Empty e t -> Execution-Proof e e'
    -> Type-Proof Empty e' t
```

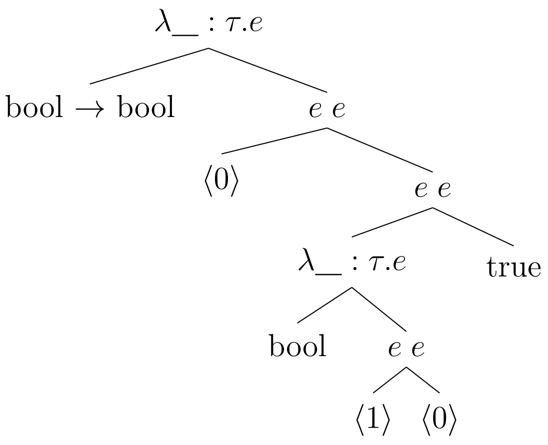
We see that the only change from the named formalization is that we explicitly require our terms to be in the empty context, but this new requirement is actually redundant because the typing judgment already enforces it.

3.4.3 Intrinsically Typed $\lambda^{\rightarrow}$ with Nameless Variables

In the third formalization, we use **intrinsic typing**. The difference is that in an intrinsically typed language, terms of the language directly encode their own proofs of well-typedness. In essence, the abstract syntax tree of the language will simultaneously serve as the type deduction tree. For example, consider the nameless lambda term from before:

$$\lambda\_ : \text{bool} \to \text{bool}.\langle 0 \rangle\ ((\lambda\_ : \text{bool}.\langle 1 \rangle\ \langle 0 \rangle)\ \text{true})$$

In AST form, this becomes:

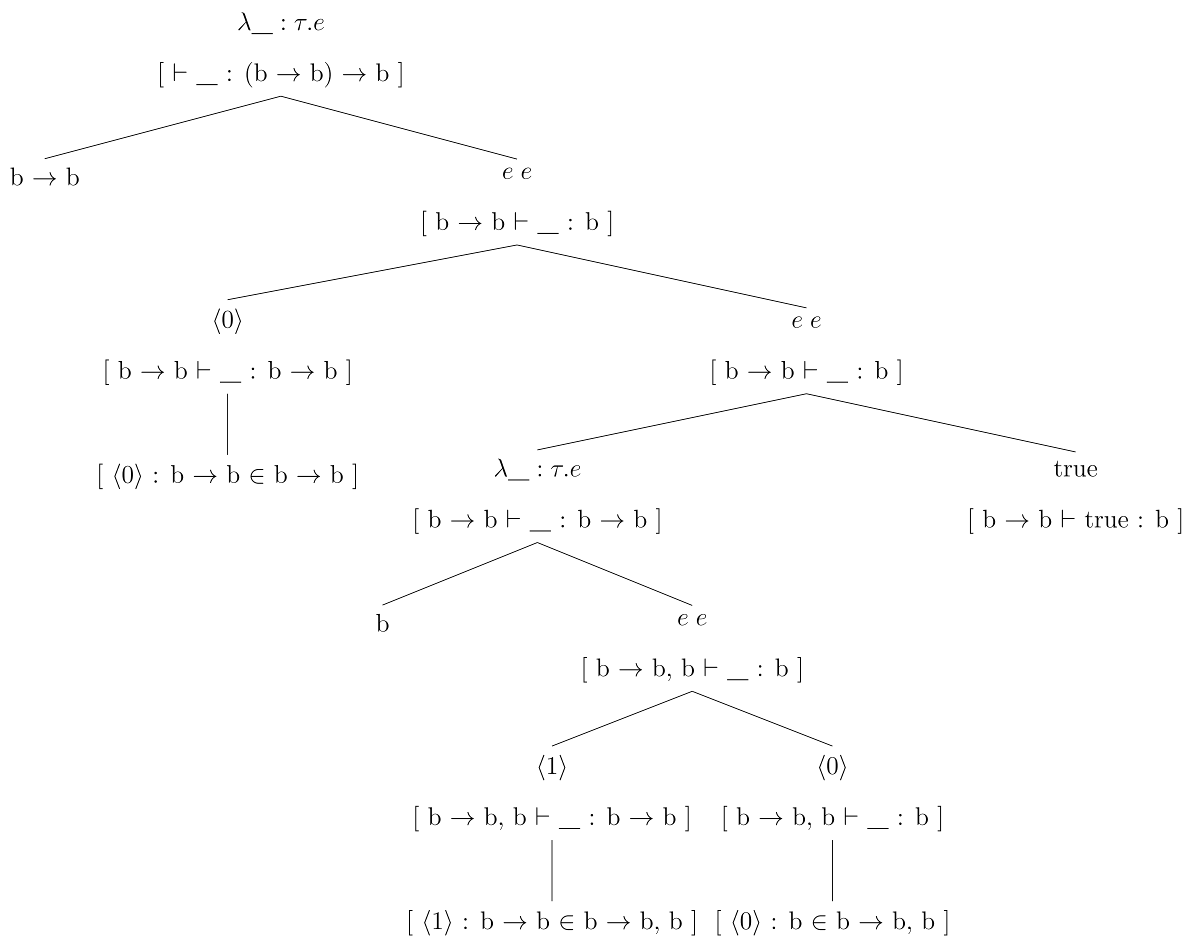


where the leaves correspond to the tokens found in the lambda term, and the parent nodes represents higher-order lambda terms that are constructed from their children. The corresponding type deduction tree is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\langle 1\rangle : \mathrm{b}\to\mathrm{b} \in \mathrm{b}\to\mathrm{b},\mathrm{b}}{\mathrm{b}\to\mathrm{b},\mathrm{b} \vdash \langle 1\rangle : \mathrm{b}\to\mathrm{b}}\,\text{T-Var} \qquad \dfrac{\langle 0\rangle : \mathrm{b} \in \mathrm{b}\to\mathrm{b},\mathrm{b}}{\mathrm{b}\to\mathrm{b}, \vdash \langle 0\rangle : \mathrm{b}}\,\text{T-Var}}{\mathrm{b}\to\mathrm{b},\mathrm{b} \vdash \langle 1\rangle\ \langle 0\rangle : \mathrm{b}}\,\text{T-App}}{\mathrm{b}\to\mathrm{b} \vdash (\lambda\_ : \mathrm{b}.\langle 1\rangle\ \langle 0\rangle) : \mathrm{b}\to\mathrm{b}}\,\text{T-Fun} \qquad \dfrac{}{\mathrm{b}\to\mathrm{b} \vdash \text{true} : \mathrm{b}}\,\text{T-True}}{\mathrm{b}\to\mathrm{b} \vdash (\lambda\_ : \mathrm{b}.\langle 1\rangle\ \langle 0\rangle)\ \text{true} : \mathrm{b}}\,\text{T-App} \qquad \dfrac{\langle 0\rangle : \mathrm{b}\to\mathrm{b} \in \mathrm{b}\to\mathrm{b}}{\mathrm{b}\to\mathrm{b} \vdash \langle 0\rangle : \mathrm{b}\to\mathrm{b}}\,\text{T-Var}}{\mathrm{b}\to\mathrm{b} \vdash \langle 0\rangle\ ((\lambda\_ : \mathrm{b}.\langle 1\rangle\ \langle 0\rangle)\ \text{true}) : \mathrm{b}}\,\text{T-App}}{\vdash \lambda\_ : \mathrm{b}\to\mathrm{b}.\langle 0\rangle\ ((\lambda\_ : \mathrm{b}.\langle 1\rangle\ \langle 0\rangle)\ \text{true}) : (\mathrm{b}\to\mathrm{b})\to\mathrm{b}}\,\text{T-Fun}$$

where b is short for bool.

To convert this term to an intrisically-typed representation, we augment each node in the AST with a typing judgment, such that the typing judgments in the children will serve as the premise for deducing the type of the parent as per the five typing deduction rules defined earlier. In intrinsically-typed AST form, the above term becomes:



where b is short for bool in the above diagram.

The advantage of using intrinsic typing over extrinsic typing is that terms that are untypable are also unrepresentable in the AST of the language. This in turn guarantees that all representable terms in the language are inherently well-typed.

To encode this formalization in Agda, we modify the data type of terms into the following:

```
data Term (Γ : Context) : Type -> Set where
    True : Term Γ Boolean
    False : Term Γ Boolean
    Var : (t : Type) (v : Variable Γ) -> (type-var Γ v ≡ t) -> Term Γ t
    Fun : (t t' : Type) -> Term (Γ , t) t' -> Term Γ (Function t t')
    App : (t t' : Type) -> Term Γ (Function t t') -> Term Γ t -> Term Γ t'
```

We see that the typing judgment of a term is present in the Agda type of the term, and the requisite premises for typing deduction are satisfied by requiring terms of the correct type as arguments to the constructors.

Since terms are inherently well-typed, there is no need for an analogous `Type-Proof` data type anymore. Values are still defined accordingly, and the definition can be found in the Appendix. The change to the definition of the evaluation semantics is subtle, but significant:

```
data Execution-Proof : (t : Type) -> Term Empty t -> Term Empty t
    -> Set where
    ...
```

We see that we can now require that evaluation steps only between terms of the same type `t`. This implies that Progress alone is sufficient to ensure type safety. Therefore, to prove type safety, we only need to define a function that satisfies the following signature:

```
Type-Safety : (t : Type) (e : Term Empty t) -> IsVal-Proof Empty t e
    ⊎ Σ [e' ∈ Term Empty t] Execution-Proof Empty t e e'
```

4 Results

To assess how each formalization of the $\lambda^{\rightarrow}$ affected the difficulty of the corresponding type safety proof, we measure the number of lines of definitions and of proofs needed to show type safety. The following are the criteria to be counted for each:

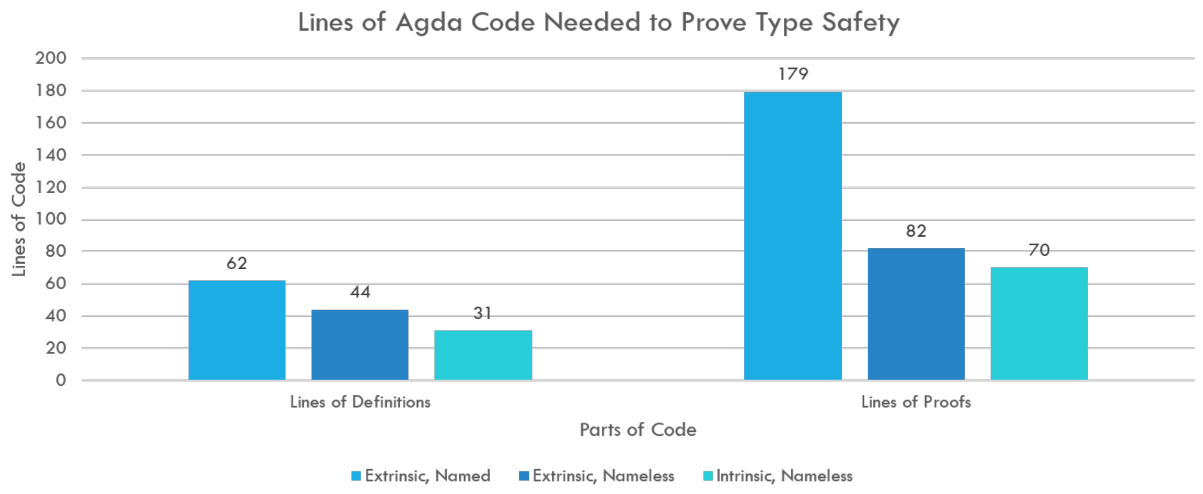


Figure 1: Number of lines of Agda code needed to prove type safety.

Definition:

1. Type definition of data types
2. Constructor definition of data types
3. Type definition of functions

Proof:

1. Body of functions

Note that we are not counting the definitions and proofs of standard library tools, such as sum types and product types, equivalence relations and their associated properties, and general deduction strategies, such as *ex falso quodlibet*. The results are shown in Figure 1. We found that using named variables drastically increased the number of lines of code needed to prove type safety for both definition and proof. We also found that using extrinsic typing requires more lines of definition proof than using intrinsic typing, although the difference is not as extreme.

5 Discussion

5.1 Named Variables vs. Nameless Variables

When going from using named variables to using nameless variables, we were able to avoid defining and proving many results related to variable shadowing when performing substitution. For example, performing the following substitution:

$$[\text{true}/x]((\lambda x : \text{bool}.x)\ x)$$

yields:

$$(\lambda x : \text{bool}.x)\ \text{true}$$

and not:

$$(\lambda x : \text{bool}.\text{true})\ \text{true}$$

This is because, by convention, we interpret the x in the body to refer to the inner binding. We refer to this phenomenon as variable shadowing. Therefore, we need to expend additional effort to account for this in our proofs. However, for nameless terms, ensuring that the corresponding substitution:

$$[\text{true}/\langle 0\rangle]((\lambda\_\_ : \text{bool}.\langle 0\rangle)\ \langle 0\rangle)$$

yields:

$$(\lambda\_\_ : \text{bool}.\langle 0\rangle)\ \text{true}$$

is much simpler because we know that in the function body, we should be replacing instances of $\langle 1 \rangle$, since a new variable has been bound when entering the body of the function.

Furthermore, the current proof of type safety for $\lambda^{\rightarrow}$ with named variables does not attempt to address the issue of variable capture. As mentioned before, in a call-by-value order of evaluation, one can exclude functions with unbound variables from the definition of values, which implies that unbound variables will never be captured during substitution. However, if we were to change the order of evaluation to full β-reduction, which will also attempt to perform substitution on all application expressions, the current proof of type safety for $\lambda^{\rightarrow}$ with named variables will not go through. In contrast, type safety proofs for $\lambda^{\rightarrow}$ with nameless variables is robust to this change because variable are never referenced ambiguously.

However, we find that using named variables aids in the human interpretability of terms of $\lambda^{\rightarrow}$. To interpret a term that uses de Brujin indices for variables, one must continuously count backward the number of steps equal to the indices to determine the semantics of the variables, which can easily become unwieldy in highly nested terms. For example, most would agree that:

$$\lambda x : \text{bool} \rightarrow \text{bool}.\lambda y : \text{bool}.(\lambda z : \text{bool}.x\ z)\ ((\lambda v : \text{bool}.\lambda w : \text{bool}.x\ v)\ y\ y)$$

is much more interpretable than:

$$\lambda\_\_ : \text{bool} \rightarrow \text{bool}.\lambda\_\_ : \text{bool}.(\lambda\_\_ : \text{bool}.\langle 2 \rangle\ \langle 0 \rangle)\ ((\lambda\_\_ : \text{bool}.\lambda\_\_ : \text{bool}.\langle 3 \rangle\ \langle 1 \rangle)\ \langle 0 \rangle\ \langle 0 \rangle)$$

One would imagine that interpreting the $\langle 0 \rangle$ at the end of the expression to be the second bound variable in the expression would require more effort than matching the name y in the example with names. Therefore, we can conclude that while named variables make proofs about programming languages more difficult, they still provide other important benefits that go beyond proofs.

5.2 Extrinsic Typing vs. Intrinsic Typing

On the other hand, when going from extrinsic typing to intrinsic typing, we see that we save having to define an external `Type-Proof` data type, as well as save a few more lines of definition in collapsing functions that would otherwise perform analogous operations on terms and typing judgments. The main source of savings for lines of proofs comes from not having to prove the Preservation theorem and its dependencies, such as the Uniqueness of Types theorem, which states that in any context, a term has at most one type.

However, similar to the case of named variables vs. nameless variables, we see that having extrinsic typing allows terms to be written down and interpreted much more easily. At the cost of being able to write down ill-typed terms, we greatly reduce the amount of effort it takes to represent a term in the language. Furthermore, the task of determining whether a term is well typed or not can be done algorithmically, using methods such as Robinson's unification algorithm [7]. Therefore, as a practical matter, it is more convenient to let human programmers write down the terms in the language, and to use a machine to derive the proper typing judgment for those terms.

6 Conclusion and Future Work

In this project, we studied three different formalizations of the simply typed lambda calculus and compared how they affected the structure and difficulty of the respective proofs of type safety. We found that type safety can be proven much more easily in a language that uses intrinsic typing and nameless variables, and while having an extrinsic typing mechanism does not complicate type safety proofs by much, using named variable can significantly increase the amount of work necessary. However, from a practical point of view, it is much better to program in a language that uses variable names and that does not require one to provide proofs of well-typedness as part of the syntax.

One possible direction to explore is to use some of the more recently developed techniques in type theory, such as **higher inductive types**, to formalize a version of $\lambda^{\rightarrow}$ that allows for named variables in the language while simultaneously avoiding many of the problems associated with variable names. A higher inductive type allows us not only to define constructors for a given type, but also to define artificial equivalences between terms of that type. For example, we can use a higher inductive type to define the terms of $\lambda^{\rightarrow}$, and then define all terms that differ by a renaming of a bound variable (often called an α-conversion) to be equivalent (i.e. α-equivalent). This will resolve the issue of variable capture during substitution, as we can simply use α-equivalence to convert the problematic terms to equivalent forms that have no variable name collision. Furthermore, since such a conversion is done in the context of an equivalence, proofs about these conversions will hopefully turn out to be relatively short because we are able to leverage properties of equivalences, such as symmetry, transitivity, congruence, etc., which we are able to use directly.

Overall, there is much more to be explored in the realm of formalization of type safety proofs. We hope to continue to explore this more.

References

[1] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):5668, 1940.

[2] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mrtberg. Cubical type theory: a constructive interpretation of the univalence axiom, 2016.

[3] Ulf Norell et al. Agda's documentation, 2019.

[4] William A Howard. The formulae-as-types notion of construction.

[5] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 9. Bibliopolis Naples, 1984.

[6] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[7] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12:23–41, 1965.

[8] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

Appendix

Detailed definitions and proofs of type safety in Agda for the three formalizations studied in this project can be found online here: `https://github.com/coolfan/cos-iw-s2019`. Please look in the following files:

1. `stlc-extrinsic.agda`: Extrinsically typed $\lambda^{\to}$ with named variables
2. `stlc.agda`: Extrinsically typed $\lambda^{\to}$ with nameless variables
3. `stlc-intrinsic.agda`: Intrinsically typed $\lambda^{\to}$ with nameless variables